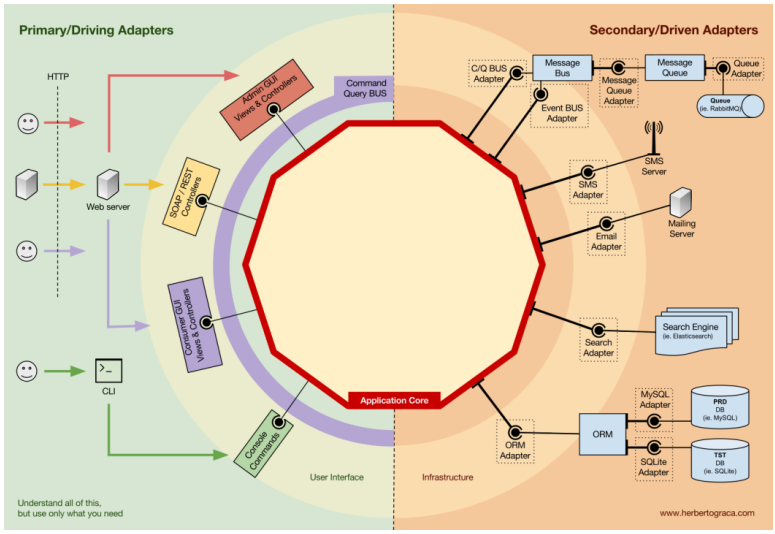


Clean Architecture - Parte 1

Michel Torres
Gerente Sênior de Tecnologia e Arquiteto de Sistemas e Soluções | Agile | Product Manager | IT Manager |...

3 de março de 2024

Visão geral

A arquitetura Clean Architecture é um estilo arquitetural que enfatiza a separação de preocupações e a manutenção da independência entre diferentes componentes de um sistema. Ela foi proposta por Robert C. Martin, também conhecido como "Uncle Bob" e se baseia nos princípios do Design Orientado a Objetos e do Desenvolvimento Orientado a Testes.

A ideia principal por trás da Clean Architecture é promover a construção de sistemas que sejam flexíveis, testáveis, e que possam evoluir ao longo do tempo sem que mudanças em um componente afetem outros desnecessariamente. Para alcançar esse objetivo, a arquitetura é dividida em camadas concêntricas, com cada camada dependendo apenas das camadas mais internas.

As camadas principais na Clean Architecture geralmente incluem:

1. **Entidades**: Contém os modelos de domínio ou entidades principais do sistema, que encapsulam o estado e o comportamento essencial do negócio.
2. **Casos de Uso (Use Cases)**: Representa as regras de negócio da aplicação. Esta camada coordena as interações entre as entidades e os detalhes técnicos da aplicação.
3. **Interfaces do Usuário (UI)**: Responsável por apresentar informações ao usuário e coletar entradas dele. Essa camada pode incluir interfaces gráficas de usuário, interfaces de linha de comando, APIs, etc.
4. **Frameworks e Drivers Externos**: Esta é a camada mais externa e contém detalhes técnicos como frameworks, bibliotecas e infraestrutura específica da plataforma (como banco de dados, web frameworks, etc.).

Destrinchando Clean Architecture

Vamos começar relembrando as arquiteturas **EBI** e **Ports & Adapters**. Ambas arquiteturas fazem uma separação explícita entre:

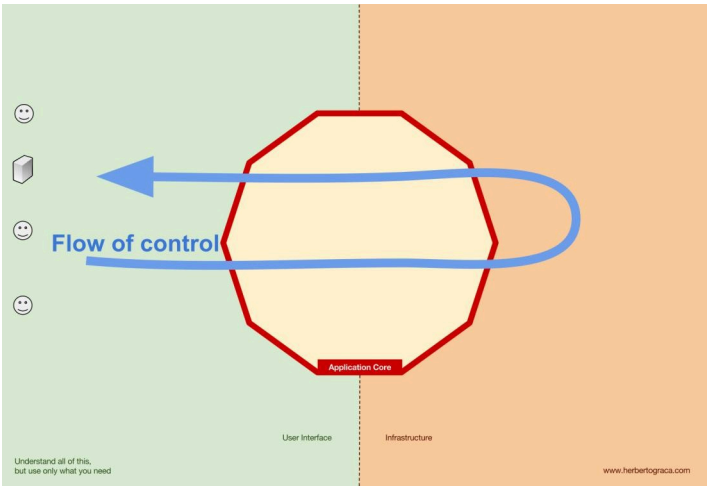
- qual código é interno da aplicação,
- o que é externo
- e o que é usado para conectar código interno e externo.

Além disso, a arquitetura **Ports & Adapters** identifica explicitamente três blocos fundamentais de código em um sistema:

- O que torna possível executar uma **interface de usuário**, seja qual for o tipo de interface de usuário;
- A **lógica de negócios** do sistema, ou **núcleo do aplicativo**, que é usada pela interface do usuário para realmente fazer as coisas acontecerem;
- **Código de infraestrutura**, que conecta o núcleo do nosso aplicativo a ferramentas como um banco de dados, um mecanismo de pesquisa ou APIs de terceiros.

O núcleo do aplicativo é o que realmente devemos nos preocupar. É o código que permite contem o objetivo da nossa aplicação. Ele pode usar várias interfaces de usuário (aplicativo web, móvel, CLI, API, ...), mas o código que realmente faz o trabalho está localizado no núcleo do aplicativo, não importa o que há a UI ou acima dela .

Como você pode imaginar, o fluxo típico do aplicativo vai do código na interface do usuário, passando pelo núcleo do aplicativo até o código da infraestrutura, de volta ao núcleo do aplicativo e finalmente, entrega uma resposta à interface do usuário.



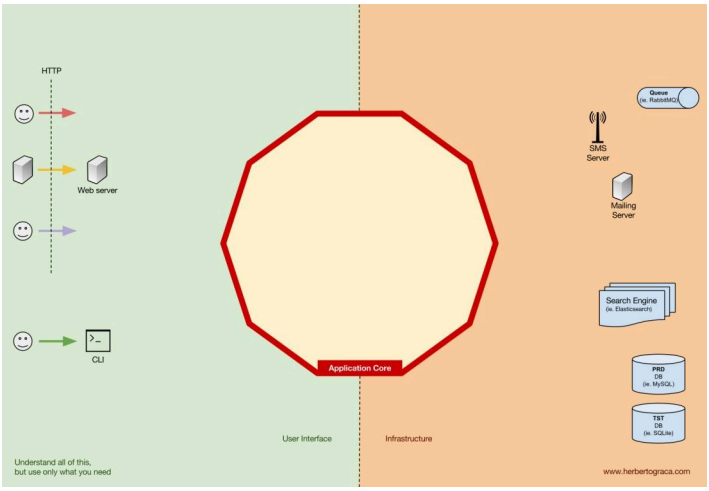
Fluxo na aplicação

Ferramentas

O núcleo da aplicação, de Longe o código mais importante, temos as ferramentas utilizadas pela aplicação, por exemplo, um motor de base de dados, regras de um motor de busca, um servidor Web e etc.

Embora possa parecer estranho, colocar uma interface UI no mesmo "balde" de um mecanismo de banco de dados e embora tenham finalidades diferentes, na verdade, são ferramentas **usadas** pela aplicação. A principal diferença é que enquanto o interface UI **instrui** nosso aplicativo o que fazer, o mecanismo de banco de dados

é **instruído**. Esta é uma distinção muito relevante, pois tem fortes implicações na forma como construímos o código que conecta essas ferramentas ao núcleo da aplicação.



Ferramentas

Conectando as ferramentas e mecanismos de entrega ao núcleo da aplicação

Adapters

As unidades de código que conectam as ferramentas ao núcleo da aplicação são chamadas de Adapters (adaptadores). Os Adapters são aqueles que implementam efetivamente o código que permitirá que a lógica de negócio se comunique com uma ferramenta específica e vice-versa.

Os Adapters que **instruem** nosso aplicativo a fazer algo são chamados de Adapters **Primários ou Condutores**, enquanto aqueles que nosso aplicativo **instrui** a fazer algo são chamados de Adapters **Secundários ou Acionados**.

Ports

Esses *Adapters*, entretanto, não são criados aleatoriamente. Eles são criados para ajustar um ponto de entrada muito específico ao núcleo da aplicação, um ***Port (uma porta)***. Um Port **nada mais é do que uma especificação** de como a ferramenta pode usar o núcleo do aplicativo ou como ela é usada pelo núcleo do aplicativo. Na maioria das linguagens e na sua forma mais simples, esta especificação, o Port, será uma Interface, mas na verdade pode ser composta por diversas Interfaces e DTOs.

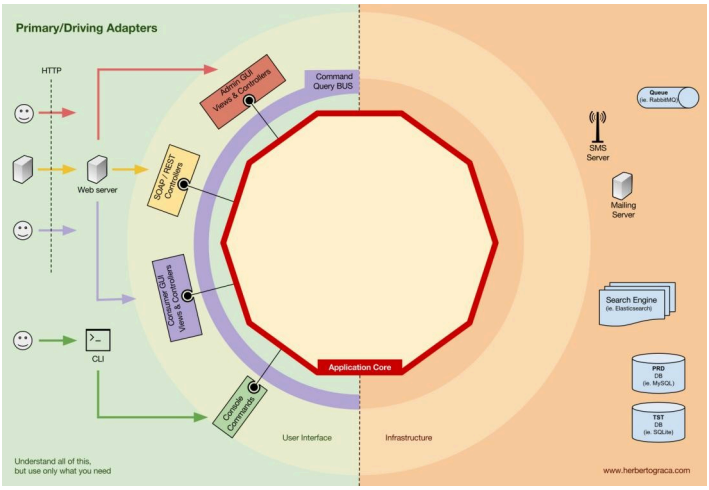
É importante observar que **as Ports (interfaces) pertencem à lógica de negócios**, enquanto os **Adapters pertencem à parte externa**. Para que esse padrão funcione como deveria, é de extrema importância que as Ports sejam criadas para atender às necessidades do núcleo da aplicação e não simplesmente imitar as APIs das ferramentas.

Adapters primários

Os Adapters Primários (de Condução) **envolvem uma porta** e a usa para informar ao núcleo do aplicativo o que fazer. Em outras palavras, nossos Adapters Primários são Controllers que são **injetados** em seu construtor com algum objeto cuja classe **implementa** a interface (Port) que o comando do controlador ou console requer.

Em um exemplo mais concreto, um Port pode ser uma interface de Serviço exigida por uma controller. A implementação concreta do Serviço é então injetada e usada no Controller.

Alternativamente, um port pode ser uma interface de barramento de comando ou barramento de consulta. Neste caso, uma implementação concreta do Barramento de Comando ou Consulta é injetada na Controller, que então constrói um Comando ou Consulta e o passa para o Barramento relevante.



Ports e Adapters Primários

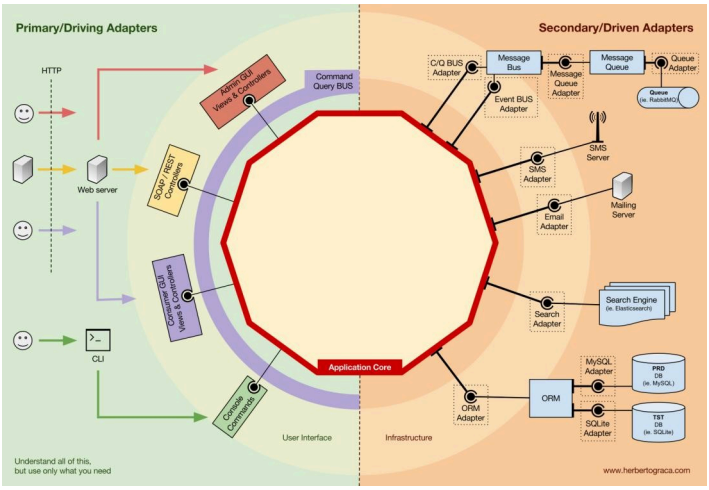
Adapters secundários

Ao contrário dos Adapters **Primários envolvem** um Port, os Adapters **Secundários implementam** um Port (uma interface) e são então injetados no núcleo da aplicação, onde quer que o Port seja necessária.

Por exemplo, vamos supor que temos um aplicativo ingênuo que precisa persistir dados. Assim criamos uma interface de persistência que atenda às suas necessidades, com um método para *salvar* um array de dados e um método para *deletar* uma linha de uma tabela pelo seu ID. A partir de então, sempre que nossa aplicação precisar salvar ou deletar dados precisaremos em seu construtor um objeto que implemente a interface de persistência que definimos.

Agora criamos um Adapter específico para MySQL que implementará essa interface. Ele terá os métodos para salvar um array e deletar uma linha em uma tabela, e iremos injetá-lo onde quer que a interface de persistência seja necessária.

Se em algum momento decidirmos mudar o fornecedor do banco de dados, digamos para PostgreSQL ou MongoDB, só precisamos criar um novo Adapte) que implemente a interface de persistência e seja específico para PostgreSQL, e injetar o novo Adapter em vez do antigo (MySQL).



Ports, Adapters Primários e Adapters Secundários

Exemplo: Fabricação de Carro

Vamos considerar um exemplo prático de Clean Architecture aplicada à fabricação de um carro.

1. **Entidades**: Dentro dessa camada, teríamos objetos que representam conceitos de negócio no contexto da fabricação de carros. Isso pode incluir entidades como Carro, Motor, Roda, etc. Essas entidades encapsulam o estado e o comportamento essencial do negócio.
2. **Casos de Uso (Use Cases)**:Aqui estarão as regras de negócio específicas da fabricação de carros. Por exemplo, poderíamos ter casos de uso como "MontarCarro", "TestarMotor", "InstalarRodas", etc. Esses casos de uso coordenam as interações entre as entidades e os detalhes técnicos do processo de fabricação.
3. **Interfaces do Usuário (UI)**: Neste exemplo, a interface do usuário pode ser tanto a interface física na fábrica, onde os trabalhadores montam o carro, quanto uma interface de software utilizada para monitorar e controlar o processo de fabricação. Esta camada se preocupa em apresentar informações sobre o progresso da fabricação, coletar dados sobre problemas encontrados durante o processo, etc. **Portas Primárias - Interface de Controle de Produção na Fábrica**: Este port permite que os funcionários da linha de produção controlem e monitorem o processo de fabricação do carro. Eles podem usar telas touchscreen, painéis de controle ou até mesmo dispositivos móveis para interagir com o sistema. **Adaptadores Primários - Controlador de Interface de Controle na Fábrica**: Este adaptador implementa a interface de controle de produção na fábrica. Ele recebe comandos dos funcionários, valida e executa as operações necessárias, como iniciar ou parar a linha de montagem, ajustar a velocidade das esteiras, etc.
4. **Frameworks e Drivers Externos**: Esta camada incluiria todos os detalhes técnicos necessários para a fabricação de carros, como máquinas de montagem, equipamentos de teste, sistemas de controle de qualidade, etc. Também podem incluir tecnologias externas, como fornecedores de peças, sistemas de gerenciamento de inventário, etc. **Adaptadores Secundários - Adaptador de Comunicação com o Sistema ERP da Fábrica**: Este adaptador é responsável por integrar o sistema de fabricação de carros com o sistema de planejamento de recursos empresariais (ERP) da fábrica.

Na prática, a separação dessas camadas permite que mudanças em uma área não afetem desnecessariamente outras partes do sistema. Por exemplo, se for necessário atualizar o software de controle de qualidade (camada de frameworks e drivers externos), isso pode ser feito sem alterar os casos de uso específicos da fabricação de carros. Da mesma forma, se

houver uma mudança nas especificações do motor (entidades), isso pode ser tratado sem impactar diretamente a interface do usuário ou os processos de montagem. Isso torna o sistema mais flexível, testável e fácil de manter ao longo do tempo.

Conclusão

O objetivo é obter um código **fracamente acoplado** e **altamente coesa**, para que as alterações sejam fáceis, rápidas e seguras de fazer, seguindo os conceitos do SOLID:

- S — Single Responsibility Principle (Princípio da responsabilidade única)
- O — Open-Closed Principle (Princípio Aberto-Fechado)
- L — Liskov Substitution Principle (Princípio da substituição de Liskov)
- I — Interface Segregation Principle (Princípio da Segregação da Interface)
- D — Dependency Inversion Principle (Princípio da inversão da dependência)

Precisamos entender esses padrões, mas também precisamos sempre pensar e entender exatamente o que nossa aplicação precisa, até onde devemos ir em prol do desacoplamento e da coesão. Essa decisão pode depender de vários fatores, começando pelos requisitos funcionais do projeto, mas também pode incluir fatores como o prazo para construir o aplicativo, a vida útil do aplicativo, a experiência da equipe de desenvolvimento e assim por diante.